

SHA

Crittografia

Luciano Margara

Unibo

2025

# SHA: Introduzione

SHA (Secure Hash Algorithm) indica una famiglia di cinque diverse funzioni hash crittografiche sviluppate a partire dal 1993 dalla NSA e pubblicato dal NIST come standard federale dal governo degli USA (FIPS PUB 180-4).

# SHA: Introduzione

SHA produce un message digest di lunghezza fissa partendo da un messaggio di lunghezza variabile. La sicurezza di un algoritmo di hash risiede nel fatto che la funzione non sia invertibile e che non deve essere mai possibile creare intenzionalmente due messaggi diversi con lo stesso digest (cioè dato  $x_1$  e  $x_2 \rightarrow f(x_1) = f(x_2)$ )

# SHA: Introduzione

Varianti SHA: SHA-1, SHA-224, SHA-256, SHA-384 e SHA-512. Le ultime 4 varianti sono spesso indicate come SHA-2. Il primo produce un digest del messaggio di soli 160 bit, mentre gli altri producono digest di lunghezza in bit pari al numero indicato nella loro sigla. L'SHA-1 è il più diffuso algoritmo della famiglia SHA ed è utilizzato in numerose applicazioni e protocolli nonostante sia ormai insicuro e verrà presto sostituito dagli altri, più moderni ed efficienti

# SHA: Introduzione

La sicurezza di SHA-1 è stata appunto compromessa dai crittoanalisti. Si veda ad esempio

[https://www.schneier.com/blog/archives/2005/02/cryptanalysis\\_o.html](https://www.schneier.com/blog/archives/2005/02/cryptanalysis_o.html)

# SHA: Introduzione

Sebbene non siano ancora noti attacchi alle varianti SHA-2, esse hanno un algoritmo simile a quello di SHA-1 per cui sono in atto sforzi per sviluppare algoritmi di hashing alternativi e migliorati. Il NIST nel 2007 ha indetto un concorso pubblico per la realizzazione di una nuova funzione SHA-3. Vincitore nel 2012: algoritmo italo-belga Keccak. Keccak sembra dunque destinato a venire gradualmente incluso e adottato nelle soluzioni di sicurezza informatica più comuni

## SHA-0 e SHA-1

La specifica originale dell'algoritmo fu pubblicata nel 1993 come Secure Hash Standard, FIPS PUB 180, dal NIST. Ci si riferisce spesso a questa versione come SHA-0 per distinguerla dalle successive versioni. Fu ritirata dall'NSA breve tempo dopo la pubblicazione e fu soppiantata da una versione rivista, pubblicata nel 1995 (FIPS PUB 180-1) e solitamente nota come SHA-1

# SHA-0 e SHA-1

L'SHA-1 differisce pochissimo dall'SHA-0. La modifica, secondo l'NSA, correggerebbe un difetto nell'algoritmo originale. L'NSA non fornì nessuna ulteriore spiegazione chiarificatrice. Sono state in seguito riportate debolezze sia nel codice dell'SHA-0 sia in quello dell'SHA-1.

L'SHA-1 pare offrire maggiore resistenza agli attacchi, a supporto di quanto dichiarato dall'NSA



# SHA-0 e SHA-1

L'SHA-1 (così come l'SHA-0) produce un digest di 160 bit da un messaggio con una lunghezza massima di  $2^{64} - 1$  bit ed è basato su principi simili a quelli usati da Rivest del MIT per MD4 e MD5

# SHA-0 e SHA-1

SHA-1 è l'algoritmo più utilizzato della famiglia SHA.

Costituisce la base di numerose applicazioni e protocolli, inclusi il TLS ed SSL, il PGP, l'SSH, l'S/MIME e l'IPsec.

L'SHA-1 è anche utilizzato in sistemi di controllo versione, come il Git, per identificare la revisione dei software e come somma di controllo per verificare l'integrità di file di grosse dimensioni

In Git, ogni commit viene identificato da un hash SHA-1. Questo hash funge da "firma digitale" del contenuto del commit. L'uso dello SHA-1 in Git non ha principalmente uno scopo di sicurezza crittografica (dato che vulnerabilità note permettono la creazione di collisioni in certi contesti), ma serve per verificare l'integrità dei dati.

# SHA-1: Notazione

↪ Questo significa che i valori vanno da 0 a  $2^{32}-1$  e non possono assumere valori negativi.

Tutte le variabili sono nel formato unsigned 32-bit

I calcoli sulle variabili vengono eseguiti modulo  $2^{32}$

↪ Durante il processo di padding, alla fine del messaggio viene aggiunta una rappresentazione della lunghezza originale del messaggio. Questa lunghezza è codificata utilizzando 64 bit, permettendo così di gestire messaggi estremamente lunghi

La lunghezza del messaggio è su 64 bits

↪ Il risultato finale dell'algoritmo SHA-1 è un hash di 160 bit (20 byte).

Il digest del messaggio è su 160 bits

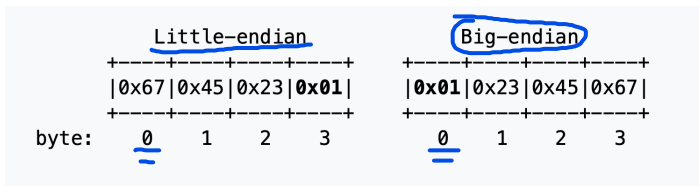
Tutte le costanti sono in formato big endian

# Little-endian vs Big-endian

Little endian e big endian sono due modalità di rappresentazione dei byte in memoria per i numeri multi-byte (come int, float, ecc.).

Big endian: il byte più significativo (MSB) viene memorizzato per primo (indirizzo più basso)


Little endian: il byte meno significativo (LSB) viene memorizzato per primo.



# SHA-1

## SHA-1(*msg*)

```
1   $\langle h_0, h_1, h_2, h_3, h_4 \rangle \leftarrow \text{INIZIALIZZA}()$ 
2   $m \leftarrow \text{PRE-PROCESSING}(msg)$ 
3   $i \leftarrow 0$ 
4   $l \leftarrow \text{length}(m)$ 
5  while  $i < l$ 
6       $chunk \leftarrow m[i \dots i + 511]$ 
7       $i \leftarrow i + 512$ 
8       $w \leftarrow \text{CHUNK-EXPANSION}(chunk)$ 
9       $\langle a, b, c, d, e \rangle \leftarrow \text{CHUNK-COMPRESSON}(w, h_0, h_1, h_2, h_3, h_4)$ 
10      $\langle h_0, h_1, h_2, h_3, h_4 \rangle \leftarrow \langle h_0, h_1, h_2, h_3, h_4 \rangle + \langle a, b, c, d, e \rangle$ 
11 return  $h_0|h_1|h_2|h_3|h_4$ 
```



Il messaggio in input viene pre-processato. In questa fase vengono applicati:

- Il padding: si aggiunge un bit "1" seguito da una serie di zeri, in modo da allineare la lunghezza del messaggio a un multiplo di 512 bit.
- L'aggiunta della lunghezza originale del messaggio, codificata su 64 bit, alla fine del messaggio.

# SHA-1

SHA-1( $msg$ )

```
1   $\langle h_0, h_1, h_2, h_3, h_4 \rangle \leftarrow \text{INIZIALIZZA}()$ 
2   $m \leftarrow \text{PRE-PROCESSING}(msg)$ 
3   $i \leftarrow 0$ 
4   $l \leftarrow \text{length}(m)$ 
5  while  $i < l$ 
6       $chunk \leftarrow m[i \dots i + 511]$ 
7       $i \leftarrow i + 512$ 
8       $w \leftarrow \text{CHUNK-EXPANSION}(chunk)$ 
9       $\langle a, b, c, d, e \rangle \leftarrow \text{CHUNK-COMPRESSION}(w, h_0, h_1, h_2, h_3, h_4)$ 
10      $\langle h_0, h_1, h_2, h_3, h_4 \rangle \leftarrow \langle h_0, h_1, h_2, h_3, h_4 \rangle + \langle a, b, c, d, e \rangle$ 
11 return  $h_0|h_1|h_2|h_3|h_4$ 
```

# SHA-1: Inizializza

Questa fase di "inizializzazione" in SHA-1 serve a definire lo stato iniziale dell'algoritmo, cioè i valori di partenza che verranno modificati iterativamente durante il processamento del messaggio.

INIZIALIZZA()

```
1   $x_0 \leftarrow 67452301$   
2   $x_1 \leftarrow EFCDAB89$   
3   $x_2 \leftarrow 98BADCFE$   
4   $x_3 \leftarrow 10325476$   
5   $x_4 \leftarrow C3D2E1F0$   
6  return  $\langle x_0, x_1, x_2, x_3, x_4 \rangle$ 
```

Punti chiave:

- Formato a 32 bit: Ogni costante è un word di 32 bit espresso in esadecimale. Questo garantisce che le operazioni successive (somme, rotazioni, ecc.) siano eseguite su dati a larghezza fissa.
- Big Endian: I valori sono scritti in formato big endian, ovvero il byte più significativo (quello con il valore maggiore) è posizionato per primo.
- Costanti Fisse: Le costanti sono predefinite e fissate dallo standard SHA-1. La loro scelta è progettata per inizializzare lo stato interno in maniera non banale e per distribuire in modo uniforme le operazioni di compressione.

# SHA-1

SHA-1( $msg$ )

```
1   $\langle h_0, h_1, h_2, h_3, h_4 \rangle \leftarrow \text{INIZIALIZZA}()$ 
2   $m \leftarrow \text{PRE-PROCESSING}(msg)$ 
3   $i \leftarrow 0$ 
4   $l \leftarrow \text{length}(m)$ 
5  while  $i < l$ 
6       $chunk \leftarrow m[i..i + 511]$ 
7       $i \leftarrow i + 512$ 
8       $w \leftarrow \text{CHUNK-EXPANSION}(chunk)$ 
9       $\langle a, b, c, d, e \rangle \leftarrow \text{CHUNK-COMPRESSION}(w, h_0, h_1, h_2, h_3, h_4)$ 
10      $\langle h_0, h_1, h_2, h_3, h_4 \rangle \leftarrow \langle h_0, h_1, h_2, h_3, h_4 \rangle + \langle a, b, c, d, e \rangle$ 
11 return  $h_0|h_1|h_2|h_3|h_4$ 
```



# SHA-1: Pre-processing

Rende la lunghezza del messaggio un multiplo di 512 bit.

Aggiunge un bit 1 alla fine del messaggio seguito da tanti 0 quanto basta per arrivare a 64 bit in meno di un multiplo di 512.

Aggiunge 64 bit che rappresentano la lunghezza originale del messaggio

- Padding con il bit "1": Dopo il contenuto originale del messaggio, viene aggiunto un singolo bit "1". Questo serve a segnare la fine del messaggio originale.

- Aggiunta di bit "0": Si aggiungono poi una serie di bit "0" fino a quando la lunghezza complessiva (messaggio originale + il bit "1" + questi bit "0") è 64 bit in meno di un multiplo di 512. L'obiettivo è che, prima di aggiungere la lunghezza del messaggio (che occupa 64 bit), il totale sia un multiplo di 512. Questo perché SHA-1 lavora su blocchi da 512 bit.

- Aggiunta della lunghezza del messaggio: Infine, vengono aggiunti 64 bit che rappresentano la lunghezza originale del messaggio (prima del padding). Questo passaggio è importante per garantire l'unicità dell'hash: la lunghezza del messaggio viene codificata e inclusa, in modo da distinguere tra messaggi diversi che potrebbero avere lo stesso contenuto nei bit ma lunghezze differenti.

# SHA-1: Pre-processing

## PRE-PROCESSING( $m$ )

```
1   $l \leftarrow \text{length}(m)$   Si calcola la lunghezza originale del messaggio  $m$ 
2   $m \leftarrow \text{append}(m, 1)$   Al messaggio  $m$  viene aggiunto un singolo bit "1"
3   $k \leftarrow \min\{i \geq 0 : (l + i + 1) \bmod 512 = 448\}$ 
4  for  $i \leftarrow 1$  to  $k$ 
5       $m \leftarrow \text{append}(m, 0)$ 
6   $m \leftarrow \text{append}(m, l)$ 
7  return  $m$ 
```

Al messaggio viene aggiunto, infine, il valore  $l$  (cioè la lunghezza originale del messaggio) codificato su 64 bit. Questa parte è fondamentale per poter recuperare il messaggio originale, verificare la sua integrità e distinguere tra messaggi che potrebbero differire solo per il padding.

Si calcola il numero minimo di bit "0" (rappresentato dalla variabile  $k$ ) da aggiungere. La condizione " $l + i + 1 \bmod 512 = 448$ " serve a garantire che, dopo aver aggiunto questi bit "0" e il bit "1" già aggiunto, la lunghezza totale del messaggio sia 64 bit in meno di un multiplo di 512. Questo spazio di 64 bit sarà poi utilizzato per apporre la lunghezza originale del messaggio.

In questo ciclo viene aggiunto un bit "0" per ogni iterazione, per un totale di  $k$  bit "0". Questi bit "0" servono a raggiungere il requisito di lunghezza specificato (448 bit prima dell'aggiunta della lunghezza).

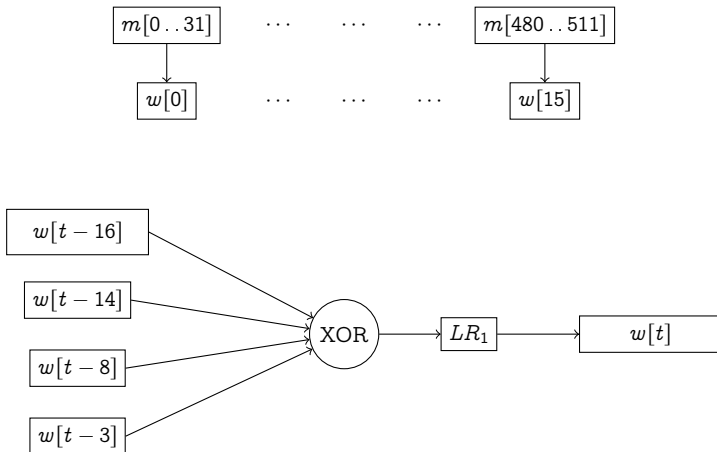
# SHA-1

SHA-1( $msg$ )

```
1   $\langle h_0, h_1, h_2, h_3, h_4 \rangle \leftarrow \text{INIZIALIZZA}()$ 
2   $m \leftarrow \text{PRE-PROCESSING}(msg)$ 
3   $i \leftarrow 0$ 
4   $l \leftarrow \text{length}(m)$ 
5  while  $i < l$ 
6       $chunk \leftarrow m[i..i + 511]$ 
7       $i \leftarrow i + 512$ 
8       $w \leftarrow \text{CHUNK-EXPANSION}(chunk)$ 
9       $\langle a, b, c, d, e \rangle \leftarrow \text{CHUNK-COMPRESSION}(w, h_0, h_1, h_2, h_3, h_4)$ 
10      $\langle h_0, h_1, h_2, h_3, h_4 \rangle \leftarrow \langle h_0, h_1, h_2, h_3, h_4 \rangle + \langle a, b, c, d, e \rangle$ 
11 return  $h_0|h_1|h_2|h_3|h_4$ 
```

Elaborazione a blocchi  
Il messaggio pre-processato viene suddiviso in blocchi (chunk) di 512 bit.  
La variabile  $i$  serve come indice per scorrere il messaggio.  
Ad ogni iterazione viene estratto un blocco (chunk) di 512 bit.

# SHA-1: Preparazione del chunk



# SHA-1: Preparazione del chunk



Da 512 bit a 2560 bit

Il passaggio "Chunk-Expansion" serve a trasformare il blocco originale di 512 bit (che è composto da 16 parole da 32 bit ciascuna) in una sequenza di 80 parole da 32 bit, per un totale di 2560 bit. Questa espansione è fondamentale per la fase di compressione dell'algoritmo, in cui ogni parola della sequenza viene utilizzata in uno specifico round di elaborazione.

## CHUNK-EXPANSION(*chunk*)

```
1  for  $i \leftarrow 0$  to 15
2       $w[i] \leftarrow chunk[32i .. 32(i + 1) - 1]$ 
3  for  $i \leftarrow 16$  to 79
4       $w[i] \leftarrow w[i - 3] \oplus w[i - 8] \oplus w[i - 14] \oplus w[i - 16]$ 
5       $w[i] \leftarrow w[i]$  leftrotate 1
6  return  $w$ 
```

Per  $i$  che va da 0 a 15, si estrae direttamente dal chunk la porzione di 32 bit corrispondente. In pratica, il chunk di 512 bit viene suddiviso in 16 parole ( $w[0]$  fino a  $w[15]$ ), dove ogni parola è una sequenza di 32 bit prelevata direttamente dal blocco.

Dopo aver calcolato tutte le 80 parole, l'algoritmo restituisce l'array  $w$  che verrà poi usato nel ciclo di compressione di SHA-1.

XOR dei word:

Si esegue la somma bit a bit (operazione XOR) delle parole in posizione  $i-3$ ,  $i-8$ ,  $i-14$  e  $i-16$ . Questa operazione combina i dati delle parole precedenti in modo non lineare, contribuendo a diffondere l'informazione.

Rotazione a sinistra:

Il risultato dell'XOR viene poi sottoposto a una rotazione a sinistra di 1 bit (leftrotate 1). Questo sposta tutti i bit di una posizione verso sinistra, con il bit più significativo che esce e ritorna come meno significativo, contribuendo ulteriormente alla dispersione e mescolamento dei bit.

# SHA-1

SHA-1(*msg*)

```
1   $\langle h_0, h_1, h_2, h_3, h_4 \rangle \leftarrow \text{INIZIALIZZA}()$ 
2   $m \leftarrow \text{PRE-PROCESSING}(msg)$ 
3   $i \leftarrow 0$ 
4   $l \leftarrow \text{length}(m)$ 
5  while  $i < l$ 
6       $chunk \leftarrow m[i..i + 511]$ 
7       $i \leftarrow i + 512$ 
8       $w \leftarrow \text{CHUNK-EXPANSION}(chunk)$ 
9       $\langle a, b, c, d, e \rangle \leftarrow \text{CHUNK-COMPRESSION}(w, h_0, h_1, h_2, h_3, h_4)$ 
10      $\langle h_0, h_1, h_2, h_3, h_4 \rangle \leftarrow \langle h_0, h_1, h_2, h_3, h_4 \rangle + \langle a, b, c, d, e \rangle$ 
11 return  $h_0|h_1|h_2|h_3|h_4$ 
```

# SHA-1: Chunk Compression

prende il blocco espanso di 80 parole (ottenute dalla Chunk-Expansion) e lo usa per aggiornare i valori interni dello stato ( $a, b, c, d, e$ ), attraverso 80 iterazioni. Ogni iterazione corrisponde a un "round" di compressione.

CHUNK-COMPRESSION( $w, a, b, c, d, e$ )

1    **for**  $i \leftarrow 0$  **to** 79

2             $\langle a, b, c, d, e \rangle \leftarrow R(w[i], i, a, b, c, d, e)$

3    **return**  $\langle a, b, c, d, e \rangle$

L'obiettivo è diffondere e confondere i dati, per rendere difficile ricostruire il messaggio originale o trovare due messaggi con lo stesso hash (proprietà crittografiche).

# SHA-1: R

$R(x, i, a, b, c, d, e)$

```
1   $\langle f, k \rangle \leftarrow F(i, b, c, d)$   
2   $temp \leftarrow (a \text{ leftrotate } 5) + f + e + k + x$   
3   $e \leftarrow d$   
4   $d \leftarrow c$   
5   $c \leftarrow b \text{ leftrotate } 30$   
6   $b \leftarrow a$   
7   $a \leftarrow temp$   
8  return  $\langle a, b, c, d, e \rangle$ 
```



# SHA-1: F

$F(i, b, c, d)$

1    **if**  $0 \leq i \leq 19$

2        **return**  $\langle (b \wedge c) \vee (\neg b \wedge d), 5A827999 \rangle$

3    **if**  $20 \leq i \leq 39$

4        **return**  $\langle b \oplus c \oplus d, 6ED9EBA1 \rangle$

5    **if**  $40 \leq i \leq 59$

6        **return**  $\langle (b \wedge c) \vee (b \wedge d) \vee (c \wedge d), 8F1BBCDC \rangle$

7    **if**  $60 \leq i \leq 79$

8        **return**  $\langle b \oplus c \oplus d, CA62C1D6 \rangle$

## SHA-1: Scelta

$$f(a, b, c) = (b \wedge c) \vee (\neg b \wedge d)$$

Se  $b$  è 1  $\rightarrow$  prendi  $c$

Se  $b$  è 0  $\rightarrow$  prendi  $d$

Quindi sceglie tra  $c$  e  $d$ , in base a  $b$

## SHA-1: Parità

$$f(b, c, d) = b \oplus c \oplus d$$

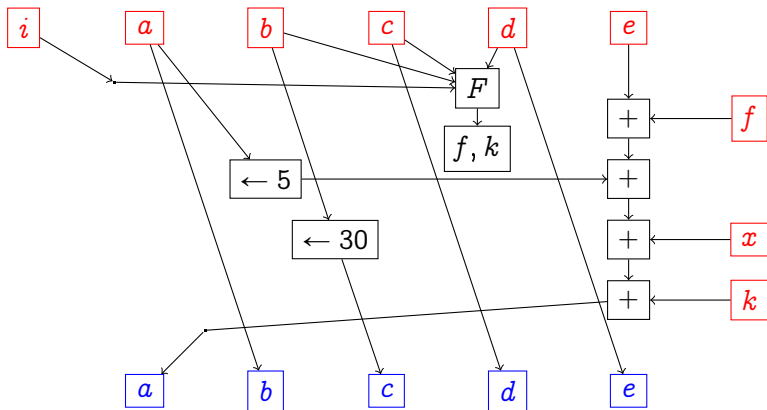
Restituisce 1 se il numero di bit a 1 tra  $b$ ,  $c$  e  $d$  è dispari

## SHA-1: Maggioranza

$$f(b, c, d) = (b \wedge c) \vee (b \wedge d) \vee (c \wedge d)$$

Restituisce 1 se almeno due dei tre bit sono 1

SHA-1:  $R(x, i, a, b, c, d, e)$



# SHA-1

SHA-1( $msg$ )

```
1   $\langle h_0, h_1, h_2, h_3, h_4 \rangle \leftarrow \text{INIZIALIZZA}()$ 
2   $m \leftarrow \text{PRE-PROCESSING}(msg)$ 
3   $i \leftarrow 0$ 
4   $l \leftarrow \text{length}(m)$ 
5  while  $i < l$ 
6       $chunk \leftarrow m[i \dots i + 511]$ 
7       $i \leftarrow i + 512$ 
8       $w \leftarrow \text{CHUNK-EXPANSION}(chunk)$ 
9       $\langle a, b, c, d, e \rangle \leftarrow \text{CHUNK-COMPRESSION}(w, h_0, h_1, h_2, h_3, h_4)$ 
10      $\langle h_0, h_1, h_2, h_3, h_4 \rangle \leftarrow \langle h_0, h_1, h_2, h_3, h_4 \rangle + \langle a, b, c, d, e \rangle$ 
11 return  $h_0|h_1|h_2|h_3|h_4$ 
```

# SHA-2

Nel 2001 il NIST pubblicò SHA-224, SHA-256, SHA-384 e SHA-512, ognuna con un digest più lungo di quello originale, collettivamente denominate SHA-2. Da notare che gli ultimi tre algoritmi furono ufficializzati come standard nel 2002 mentre l'SHA-224 fu introdotto nel febbraio del 2004: quest'ultimo presenta un hash di lunghezza identica a quella di 2 chiavi del Triple DES.

# SHA-2

Tutte queste varianti sono brevettate dal governo statunitense, ma pubblicate con licenza libera. Gli algoritmi SHA-256 e SHA-512 lavorano, rispettivamente, con word di 32 e 64 bit. La loro struttura è sostanzialmente identica. Gli algoritmi SHA-224 e SHA-384 sono semplicemente versioni troncate dei precedenti due, con hash calcolati con differenti valori iniziali. Gli algoritmi SHA-2 non hanno ricevuto, a differenza dell'SHA-1, molta attenzione dalla comunità dei crittoanalisti per cui la loro sicurezza in campo crittografico non è stata del tutto provata



## SHA-256: Inizializza

Primi 32 bits della parte frazionaria della radice quadrata  
dei primi 8 numeri primi

INIZIALIZZA()

```
1   $x_0 \leftarrow 6a09e667$   
2   $x_1 \leftarrow bb67ae85$   
3   $x_2 \leftarrow 3c6ef372$   
4   $x_3 \leftarrow a54ff53a$   
5   $x_4 \leftarrow 510e527f$   
6   $x_5 \leftarrow 9b05688c$   
7   $x_6 \leftarrow 1f83d9ab$   
8   $x_7 \leftarrow 5be0cd19$   
9  return  $\langle x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7 \rangle$ 
```

# SHA-256: Costanti

Primi 32 bits della parte frazionaria della radice cubica  
dei primi 64 numeri primi

```
k[0..63] :=  
0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,  
0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,  
0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,  
0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,  
0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,  
0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,  
0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,  
0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2
```

# SHA-256: Pre-processing

PRE-PROCESSING( $m$ )

1  $l \leftarrow \text{length}(m)$

2  $m \leftarrow \text{append}(m, 1)$

3  $k \leftarrow \min\{i \geq 0 : (l + i + 1) \bmod 512 = 448\}$

4 **for**  $i \leftarrow 1$  **to**  $k$

5      $m \leftarrow \text{append}(m, 0)$

6  $m \leftarrow \text{append}(m, l)$

7 **return**  $m$

# SHA-256: Preparazione del chunk

CHUNK-EXPANSION(*chunk*)

```
1  for  $i \leftarrow 0$  to 15
2       $w[i] \leftarrow \text{chunk}[32i .. 32(i+1) - 1]$ 
3  for  $i \leftarrow 16$  to 63
4       $s_0 \leftarrow (w[i-15] \text{ rightrotate } 15) \oplus$   

        $(w[i-15] \text{ rightrotate } 18) \oplus (w[i-15] \text{ rightshift } 3)$ 
5       $s_1 \leftarrow (w[i-2] \text{ rightrotate } 17) \oplus$   

        $(w[i-2] \text{ rightrotate } 19) \oplus (w[i-2] \text{ rightshift } 10)$ 
6       $w[i] \leftarrow w[i-16] + s_0 + w[i-7] + s_1$ 
7  return  $w$ 
```

# SHA-256: F

$F(x, y, i, a, b, c, e, f, g, h)$

- 1  $s_0 \leftarrow (a \text{ rightrotate } 2) \oplus$   
 $(a \text{ rightrotate } 13) \oplus (a \text{ rightrotate } 22)$
- 2  $maj \leftarrow (a \wedge b) \oplus (a \wedge c) \oplus (b \wedge c)$
- 3  $t_2 \leftarrow s_0 + maj$
- 4  $s_1 \leftarrow (e \text{ rightrotate } 6) \oplus$   
 $(e \text{ rightrotate } 11) \oplus (e \text{ rightrotate } 25)$
- 5  $ch \leftarrow (e \wedge f) \oplus ((\neg e) \wedge g)$
- 6  $t_1 \leftarrow h + s_1 + ch + x + y$
- 7 **return**  $\langle t_1, t_2 \rangle$

# SHA-256: R

$R(x, y, i, a, b, c, d, e, f, g, h)$

- 1  $\langle t_1, t_2 \rangle \leftarrow F(x, y, i, a, b, c, e, f, g, h)$
- 2  $h \leftarrow g$
- 3  $g \leftarrow f$
- 4  $f \leftarrow e$
- 5  $e \leftarrow d + t_1$
- 6  $d \leftarrow c$
- 7  $c \leftarrow b$
- 8  $b \leftarrow a$
- 9  $a \leftarrow t_1 + t_2$
- 10 **return**  $\langle a, b, c, d, e, f, g, h \rangle$

# SHA-256: Chunk Compression

CHUNK-COMPRESSION( $w, k, a, b, c, d, e, f, g, h$ )

1   **for**  $i \leftarrow 0$  **to** 63

2          $\langle a, b, c, d, e, f, g, h \rangle \leftarrow R(w[i], k[i], i, a, b, c, d, e, f, g, h)$

3   **return**  $\langle a, b, c, d, e, f, g, h \rangle$

# SHA-256

SHA-256(*msg*)

```
1   $k[0..63] \leftarrow$  first 32 bits of the fractional part ...
2   $\langle h_0, \dots, h_7 \rangle \leftarrow \text{INIZIALIZZA}()$ 
3   $m \leftarrow \text{PRE-PROCESSING}(msg)$ 
4   $i \leftarrow 0$ 
5   $l \leftarrow \text{length}(m)$ 
6  while  $i < l$ 
7       $chunk \leftarrow m[i..i + 511]$ 
8       $i \leftarrow i + 512$ 
9       $w \leftarrow \text{CHUNK-EXPANSION}(chunk)$ 
10      $\langle a, \dots, h \rangle \leftarrow \text{CHUNK-COMPRESSION}(w, k, h_0, \dots, h_7)$ 
11      $\langle h_0, \dots, h_7 \rangle \leftarrow \langle h_0, \dots, h_7 \rangle + \langle a, \dots, h \rangle$ 
12 return  $h_0|h_1|h_2|h_3|h_4|h_5|h_6|h_7$ 
```



# SHA-1: Sensibilità alla condizioni iniziali

```
tot = 0;
For[i = 1, i ≤ 1000, i++,
  rw = RandomWord["KnownWords", 10 000];
  txt = ToString[rw];
  txt1 = StringReplacePart[txt, "*", {100, 100}];
  StringLength[txt];
  h1 = Hash[txt, "SHA", "HexString"];
  h2 = Hash[txt1, "SHA", "HexString"];
  t = Table[If[StringTake[h1, {i, i}] = StringTake[h2, {i, i}], 1, 0], {i, 1, StringLength[h1]}];
  tot = tot + Total[t] / 40;
];
tot / 1000 // N
```

0.060425

# SHA256: Sensibilità alla condizioni iniziali

```
tot = 0;
For[i = 1, i ≤ 1000, i++,
  rw = RandomWord["KnownWords", 10 000];
  txt = ToString[rw];
  txt1 = StringReplacePart[txt, "*", {100, 100}];
  StringLength[txt];
  h1 = Hash[txt, "SHA256", "HexString"];
  h2 = Hash[txt1, "SHA256", "HexString"];
  t = Table[If[StringTake[h1, {i, i}] == StringTake[h2, {i, i}], 1, 0], {i, 1, StringLength[h1]}];
  tot = tot + Total[t] / 64;
];
tot / 1000 // N
```

0.063125

# SHA512: Sensibilità alla condizioni iniziali

```
tot = 0;
For[i = 1, i ≤ 1000, i++,
  rw = RandomWord["KnownWords", 10 000];
  txt = ToString[rw];
  txt1 = StringReplacePart[txt, "*", {100, 100}];
  StringLength[txt];
  h1 = Hash[txt, "SHA512", "HexString"];
  h2 = Hash[txt1, "SHA512", "HexString"];
  t = Table[If[StringTake[h1, {i, i}] == StringTake[h2, {i, i}], 1, 0], {i, 1, StringLength[h1]}];
  tot = tot + Total[t] / 128;
];
tot / 1000 // N
```

0.0617422

# MD5: Sensibilità alle condizioni iniziali

```
tot = 0;
For[i = 1, i ≤ 1000, i++,
  rw = RandomWord["KnownWords", 10 000];
  txt = ToString[rw];
  txt1 = StringReplacePart[txt, "*", {100, 100}];
  StringLength[txt];
  h1 = Hash[txt, "MD5", "HexString"];
  h2 = Hash[txt1, "MD5", "HexString"];
  t = Table[If[StringTake[h1, {i, i}] == StringTake[h2, {i, i}], 1, 0], {i, 1, StringLength[h1]}];
  tot = tot + Total[t] / 32;
];
tot / 1000 // N
```

0.0619375

# Stringhe random

Due caratteri esadecimali random hanno probabilità di essere uguali pari a  $p = 1/16$

Il valore medio di caratteri uguali in stringhe esadecimali lunghe  $n$  (distribuzione binomiale) è pari a  $n p q = 0.0585938 n$

Le funzioni hash riescono a trasformare piccolissime modifiche sull'input in differenze "random" sul digest.